Decidable Languages

Reading

Section 4.1

Practice problems (page 159):

Challenge:

Decidable Languages

We will now discuss a number of decidable problems, that we have largely encountered before. But before we do so we need to discuss the inputs to our Turing Machines.

String representations of objects

We need a way to turn the objects we are interested in working with into some strings, which can then be:

1. considered as elements of a language L
2. considered as the input to a Turing Machine

The key observation was that there might be many different representations of a type of object, what is important is that we fix one, and that it is well defined and specified.

> An **encoding** or **string representation** of a class of objects is an association of a string to each object of this class so that:
>
> - distinct objects will result in distinct strings, and
> - we can describe an algorithm for determining if a given string is a representation of an object under this encoding
>
> The string corresponding to an object G under such an encoding will be denoted by $\langle G \rangle$.

All encodings for the same object would be fundamentally interchangeable. What is important is that we fix one encoding.

We briefly discussed a number of such encoding for graphs:

- Listing all edges as pairs of numbered vertices
- Listing for each vertex the set of vertices connected to it
- Filling out the full $n \times n$ adjacency matrix for the graph

- And many more, as well as variations of the above by using e.g. brackets instead of parentheses.

Of particular interest for us will be encodings of the kinds of objects we have encountered so far. For instance imagine the encoding of a DFA:

- It would be a list of 5 items
- The first item is a set of states. We can imagine it being a comma-separated list of strings like `q1`, `q2` and so on. Or if we are willing to think of our states as the sequential integers 1, 2, 3, and so on, we could store a single number of how many states we have.
- The second item will be a list of the allowed alphabet symbols. This is distinct from the alphabet for our Turing Machine, which is thought to be richer.
- The third item is a transition function. This is a list of triples like `(1,a,3)` indicating the current state, an input and then the state to transition to.
- The fourth item will be a single string corresponding to the start state.
- The fifth item will be a list of the accept states.

So given a DFA we can convert it to the above string. Conversely, given a string we can determine if it has the form specified above, i.e. we can write a Turing Machine that would do each of the following:

- Needs to be a list of 5 items.
- The first and second items need to themselves be lists.
- The third item needs to be a list of triples, where the first entry and the third entry in a triple need to be from the states that were in the first item, and the second entry needs to be from the list of alphabet symbols that were in the second item.
- The fourth item needs to be one of the states listed in the first item.
- Each of the entries in the fifth item need to be amongst the states listed in the first item.

Each of these steps would be tedious to write in a TM but perfectly doable.

Exercise: I purposefully omitted something from the checks that need to happen. What is it?

Exercise: We can imagine a similar process for an NFA. What would change?

Decidable Languages related to DFAs/NFAs

We can now consider some languages related to these DFAs. We can for instance imagine the language:

$$L = \{\langle \text{A} \rangle \mid A \text{ is a DFA}\}$$

Essentially the steps we described above tell us that this language L is decidable: Given an arbitrary string, we can tell if it has the form specified above.

Here is a more interesting language:

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts on input string } w\}$$

So this is the language of all pairs of a DFA and an input string, such that the DFA accepts the string.

Asking that this language be decidable is the same thing as asking: Can we write an algorithm that given a DFA and an input string can decide if the DFA accepts the string or not? We will call this the **acceptance problem** for DFAs.

The answer is obviously yes: We run the DFA on the string, and see what ends up happening.

But let us be a bit more precise in our use of the term "run":

- Use a 3-tape Turing Machine.
- Given the input $\langle B, w\rangle$, we first make sure it is of the correct format (B has to be a DFA).
- We write the string w into the second tape, and write/keep the DFA B in the first tape.
- We find the start state and write it into the third tape.
- We then simulate the DFA operations:

  1. We look at the current state in the third tape.
  2. We also look at the head of the second tape ("next" character of the input string).
  3. If that character is blank, we have processed all input.
     - Look through the list of final states to see if any of them matches the state in the third tape. If one of them does then we accept, if it does not we reject. In any case, if the character is blank, we terminate.
  4. We look into the DFA to the transition function list, and find what is supposed to happen on that input and that state.
  5. We write this state into the third tape.
  6. We advance the second tape's head to the next character.
  7. We go back to step 1.

In the future we will abbreviate this even more to simply say that we "simulate B on input w".

It is clear by part 3 above that this Turing Machine is a decider: It will terminate when the tape 2 reaches the end of the string w. So we have:

> The language A_{DFA} is Turing-decidable. In other words, the problem of determining if a DFA accepts a certain string is a decidable problem.

Exercise: What about the language A_{NFA} that does the same thing but for an NFA?

We could model this perhaps via a deterministic Turing Machine, but there is a simple way: Convert the NFA to a DFA. So the construction might go something like this:

- Given an input $\langle N, w \rangle$ where N is an NFA.
- Construct the DFA D that is equivalent to N.
- Call upon the TM that detects if D accepts on input w. Return its result.

This is an important idea: We reduce our problem to a previously solve problem. We use previously constructed TMs as "subroutines", or if you prefer as helper functions which we can call on to do part of the work.

> Exercise: Consider the language:
>
> $$E_{\text{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) \neq \emptyset\}$$
>
> In other words, the language contains all DFAs that have at least one string that they accept. Is this language decidable? Can you describe the Turing Machine that decides it? We will call this the **emptyness problem** for DFAs.
>
> Exercise: Consider the language:
>
> $$EQ_{\text{DFA}} = \{\langle A, B \rangle \mid A, B \text{ are DFAs and } L(A) = L(B)\}$$
>
> In other words the input to this language is a pair of DFAs that both recognize the same language. Is this language decidable? We will call this the **equivalence problem** for DFAs.
>
> Hint: Try to use the answer to the previous problem.

Decidable Problems concerning CFLs

Let us now consider problems related to context-free languages, and specifically context-free grammars.

Exercise: How would we represent a CFG as a string?

We can ponder the same questions as before:

- Is the acceptance problem for CFGs decidable?
- Is the emptyness problem for CFGs decidable?
- Is the equivalence problem for CFGs decidable?

We will discover that the answer to the first two questions is Yes, but the answer to the third is no:

> There is no TM that can decide, given two CFGs, whether they recognize the same language or not.

Exercise: Look back at your proof of why the equivalence problem for regular languages/DFAs was decidable. What goes wrong when we try to apply the same method to CFGs/CFLs?

Exercise: It is easy to see that the language $L = \{\langle A, B\rangle \mid a, b \text{ are CFGs such that } L(A) \neq L(B)\}$ is Turing-recognizable. How would that go?

We now proceed to discuss the other two problems:

> The acceptance problem for CFGs is decidable.

So, given a CFG and a string w, we need to decide if the CFG can generate that string.

A first idea is to try all derivations. But there are effectively infinitely many possible derivations. We need a way to stop. Otherwise what we have is a recognizer, not a decider.

The idea comes from Chomsky Normal forms:

> If a grammar is in Chomsky Normal Form, and we have a string w of length n produced by the grammar, then the derivation contains exactly $2n - 1$ steps.

This follows from the rules of the Chomsky Normal form. Think of some small values of n and it will become clear.

So now this gives us a way to deal with the acceptance problem:

- Given a CFG and a string.
- Convert the CFG to Chomsky Normal Form.
- Check only those derivations in this new CFG consisting of 2n-1 steps.
- If you find the string in one of those, accept. If you do not, reject.

This has profound implications for computing: All programming languages are codified in terms of a CFG specifying the valid syntax for programs in a language. In order to run any program in a programming language, a programming being essentially just a string of input from some alphabet, we need to parse it according to the CFG from the language. The fact that the acceptance problem is decidable tells us that this is indeed doable. Although the method described above would likely be too slow to be of practical use.

Before moving on, let us consider an important consequence of this result:

> Every CFL is decidable.

This essentially follows what we already did: Suppose that G is a CFG for the language L. Then for a string w we run the acceptance algorithm for CFGs on the pair $\langle G, w\rangle$, and we return the result of that algorithm.

This enables us to build a hierarchy of languages: The collection of all regular languages is contained within the collection of CFLs which then is contained within the collection of Turing-decidable languages, which in turn is contained in the collection of Turing-recognizable languages. All these inclusions are strict.

Now we move on to the other decidable problem:

> The emptyness problem for CFGs is decidable.

Essentially the question boils down to this: Can the start variable generate a string of literals by means of the production rules?

We will actually discuss the most general problem:

> There is an algorithm that given a CFG and a nonterminal in it, determines if that terminal can generate a string of literals via a derivation in the grammar.

Think about how this algorithm would go before reading on.

Here is how the Turing Machine might go:

- Keep a list of all nonterminals and terminals on a separate tape.
- Start by marking all terminals in that list.
- Repeat the following steps until there is no new marking produced by them:
  - Go through each production rule in the language.
  - If the RHS of that production rule consists entirely of marked items, then also mark the nonterminal on the LHS.
- Check if the nonterminal you wanted to examine is marked. Marked nonterminals are exactly those that have the potential of producing a string of terminals. In our specific problem, the nonterminal we would check is the start variable.